Notes on type-driven development with Idris

Christiano Braga

August, 2019

Universidade Federal Fluminense

Introduction

Contact information

Christiano Braga
Associate Professor

Instituto de Computação
Universidade Federal Fluminense
cbraga@ic.uff.br
http://www.ic.uff.br/~cbraga
Lattes Curriculum Vitae

Context i

- Current distributed applications ecosystem: IOT, Cloud, Web...
- A common problem in distributed information systems: *SQL code injection*.
  - Examples: Sony in 2011 and Yahoo! in 2012.
  - Losses of millions of dollars

The problem, by example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = "
        + txtUserId;
```

If `txtUserId` is equal to `105 OR 1=1`, which is always true, a malicious user may access *all* user information from a database.

Solutions i

- SQL parameters: additional values are passed to the query.
- Escaping functions: they transform the input string into a "safe" one before sending it to the DBMS.
- The problem with the solutions is that communication relies on *strings*.
- What if we could **type** this information?

Protocols i

- Web programming invariably requires following certain **protocols**.
  - For example, to connect to make a query:
    1. Create a connection.
    2. Make sure the connection was established.
    3. Prepare an SQL statement.
    4. Make sure that variables are bound.
    5. Execute the query.
    6. Process the result of the query.
    7. Close connection.
- Of course, a function could implement such a sequence, but how could one make sure that such a sequence is *always* followed?

Protocols ii

- In other words, what if we could *type* protocol behavior and make sure our Web programs *cope* with such types?
- Moreover, what if we could define special *notation* to create instances of such types?
- Protocols are one example but note that *business processes* may be treated the same way.

Service-oriented web development model i

Services are blackboxes, are stateless, are composable, among other nice characteristics.

- Services are first-class citizens in Cloud PaaS, and other platforms.
- These characteristics allow for a *clean* and *simple* interpretation of services as *functions*.
- ***What about capturing a company's way of developing PaaS as DSL?***
- ***What about capturing a company's clients processes as DSL?***

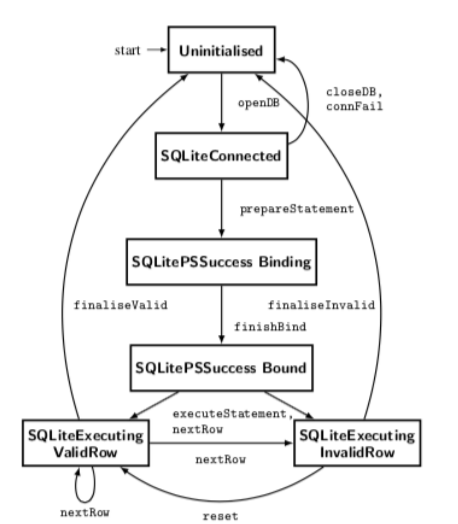
An example DSL

(From Fowler&Brady13.)

- Think of each step of a Web application as a business process.
- The notion of a Web application is typed, and so are its steps.
- For example, a Web application has forms and its forms have handlers.
- A particular Web application is *safe* (or well-typed) if its forms are well-typed. A form is well-typed if its handlers are also well-typed.

An example DSL ii

- The database protocol can be captured as a type.



An example DSL iii

- For example, the step `SQLiteConnected` step has type

```
data SQLiteConnected : Type where
     SQLConnection : ConnectionPtr -> SQLiteConnected
```

- The DSL has constructions for defining typed form handlers such as

```
handleRequest : CGIProg
     [SESSION (SessionRes SessionUninitialised),
      SQLITE ()] ()
```

that will only handle a request on properly established sessions.

Programming languages support for DSL development

- Essentially, there are two approaches for DSL-based development:
  1. Transformational approach:
     DSL program $\xrightarrow{\text{parsing}}$ Protocol data type instance $\xrightarrow{\text{transformation}}$ Web (micro)service framework.
  2. Embedded DSL approach:
     The programming languages has support the definition of notation and typing.
- Programming languages that support approach #i are Racket and Maude.
- Programming languages that support approach #ii are Idris, Lean and Haskell.

Our research approach

- To program services with domain-specific languages, implemented on top of strongly typed functional languages.
- To develop and apply program analysis techniques to DSL-based approaches to Web development.
- More specifically, to develop Web programming support in Idris.

Summing up

- We have chosen an important technical problem in web development (SQL injection), that may cause loss of millions of dollars, to illustrate DSL with functional programming usefulness.
- The issues raised here may be moved to a higher level of abstraction to represent buiness processes and their refinement into code.
- There is off the shelf technology to support this approach.

This short-course

- In this short-course we will address some of the basic concepts of the type-driven approach that gives support to the development scenario outlined in this section.

Suggested reading

Edwin Brady. 2017. Type-driven development. Manning.

Simon Fowler and Edwin Brady. 2013. Dependent Types for Safe and Secure Web Programming. In Proceedings of the 25th symposium on Implementation and Application of Functional Languages (IFL '13). ACM, New York, NY, USA, Pages 49, 12 pages. DOI: https://doi.org/10.1145/2620678.2620683

The need for types

The need for types

- This section motivates the use of strong typing with a very very simple example: Bhaskara's theorem.
- In a tutorial way, we illustrate how types are necessary and, more specifically, how Idris' strong-typing presents itself as a powerful development tool.

Bhaskara's theorem

- From school: Bhaskara's theorem[1]

$$ax^2 + bx + c = 0 \quad \Rightarrow \quad x = \frac{-b \overset{+}{-} \sqrt{\delta}}{2a}$$
$$\text{where } \delta = b^2 - 4acb$$

[1] For solving 2^{nd} degree polynomials. But this could might as well be an Excel formula, for instance! I mention Excel because that Microsoft is devoting serious efforts to develop a type system for Excel.

As functions

$$
\begin{aligned}
\texttt{bhask}(a, b, c) = & \\
& (-b + \sqrt{\texttt{delta}(a, b, c)}/2a, \\
& -b - \sqrt{\texttt{delta}(a, b, c)}/2a) \\
\texttt{delta}(a, b, c) = & \; b^2 - 4acb
\end{aligned}
$$

First attempt: no types i

- In Python:

```
from math import sqrt
def delta(a,b,c):
  return  (b * b) - (4 * a * c)
def bhask(a,b,c):
  d = delta(a,b,c)
  sr = sqrt(d)
  r1 = (-b + sr) / 2 * a
  r2 = (-b - sr) / 2 * a
  return (r1, r2)
```

First attempt: no types ii

- When we run `bhask(1,2,3)` the following is spit out:

```
Traceback (most recent call last):
  File "bhask.py", line 16, in <module>
    bhask(1,2,3)
  File "bhask.py", line 9, in bhask
    sr = sqrt(d)
ValueError: math domain error
```

- This cryptic answer is only because we rushed into a direct implementation and forgot that `delta(a,b,c)` may return a *negative* value!

Second attempt: still no types. i

- Now, assuming we are interested only on Real results, how should `bhask` deal with the possibility of a negative `delta`?
- One possibility is to raise an *exception*:

```
from math import sqrt
def delta(a,b,c):
  return  (b * b) - (4 * a * c)
def bhask(a,b,c):
  d = delta(a,b,c)
  if d >= 0:
        sr = sqrt(d)
        r1 = (-b + sr) / 2 * a
        r2 = (-b - sr) / 2 * a
```

```
        return (r1, r2)
    else:
        raise Exception("No Real results.")
```

- This implementation gives us a more *precise* answer:

```
Tue Jul 30@17:18:02:sc$ python3 -i bhask.py
Traceback (most recent call last):
  File "bhask.py", line 16, in <module>
    bhask(1,2,3)
  File "bhask.py", line 14, in bhask
    raise Exception("No Real results.")
Exception: No Real results.
```

- A very **important** point here is that we only find all this out while actually *running* our implementation. Can't we do better? That is, let the **compiler** find out that `delta` may become a negative number and complain if this is not properly handled?

Third attempt: Idris. i

- Let us play with `delta` first.
- Strongly-typed languages, such as Idris, force us to think about types right away as we need to define `delta`'s signature. If we make the same mistake we did in the first attempt and forget that `delta` may become negative, we may write,

```
> delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Nat
> delta a b c = (b * b) - (4 * a * c)
```

the compiler would tell us:

Third attempt: Idris. ii

```
Type checking ./intro.lidr
intro.lidr:100:26:
    |
100 | > delta a b c =  (b * b) - (4 * a * c)
    |                          ^
When checking right hand side of delta with expected type
        Nat
When checking argument smaller to function Prelude.Nat.-:
     Can't find a value of type
     LTE (mult (plus a (plus a (plus a (plus a 0)))) c)
     (mult b b)
```

Third attempt: Idris. iii

- This is cryptic, in a first-glance, but tells us precisely **what** is wrong **and** at **compile** time. The problem is **with subtraction**: the type checker was not able to solve the inequality, defined in Idris' libraries,

  $$4ac \leq b^2$$

  in order to produce a **natural** number while computing `delta`, as natural numbers can not be negative!

First fix. i

- And we have not even started thinking about `bhask` yet! But let us first make `delta` type right by changing its signature:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (b * b) - (4 * a * c)
```

- To see the effect of this change, load `delta-fix.lidr` with the command:

```
:l delta-fix.lidr
```

- Don't be so happy though! This is not what we want yet.

First fix. ii

```
Type checking ./delta-fix.lidr
delta-fix.lidr:5:18-38:
  |
5 | > delta a b c =  (b * b) - (4 * a * c)
  |                  ~~~~~~~~~~~~~~~~~~~~~
When checking right hand side of delta with expected type
        Int

Can't disambiguate since no name has a suitable type:
        Prelude.Interfaces.-, Prelude.Nat.-

Holes: Main.delta
```

First fix. iii

- Idris does not know which subtraction operation to use because we are operating operating with natural numbers but we should return an integer! A casting is in order!

Second fix. i

- Think about why we should cast the right-hand side expression in the following way:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (cast (b * b)) - (cast (4 * a * c))
```

and not the whole right-hand side of `delta` at once. - To see the effect of this change, load `delta-fix2.lidr` with the command:

```
:l delta-fix2.lidr
```

- You should finally be able to see

```
Type checking ./delta-fix2.lidr
*delta-fix2>
```

Second fix. ii

and run `delta 1 2 3`, for instance, to see the following result.

```
*delta-fix2> delta 1 2 3
-8 : Int
```

The road so far i

Your session should look like this at this point:

```
Mon Aug 05@14:24:16:the-need-for-types$
idris --nobanner tnft.lidr
Type checking ./tnft.lidr
tnft.lidr:107:25:
    |
107 | > delta a b c = (b * b) - (4 * a * c)
    |                         ^
When checking right hand side of delta with expected type
        Nat

When checking argument smaller to function Prelude.Nat.-:
```

The road so far ii

```
      Can't find a value of type
            LTE (mult (plus a (plus a (plus a (plus a 0))))
               (mult b b)

   Holes: Main.delta
   *tnft> :l delta-fix.lidr
   Type checking ./delta-fix.lidr
   delta-fix.lidr:5:18-38:
     |
   5 | > delta a b c =  (b * b) - (4 * a * c)
     |                  ~~~~~~~~~~~~~~~~~~~~~
   When checking right hand side of delta with expected type
           Int
```

```
Can't disambiguate since no name has a suitable type:
        Prelude.Interfaces.-, Prelude.Nat.-

Holes: Main.delta
*delta-fix> :l delta-fix2.lidr
*delta-fix2> delta 1 2 3
-8 : Int
```

Bhaskara at last! i

- Painful, no?
  No!
- The compiler is our *friend* and true friends do not always bring us good news!
- Think about it using this metaphor: do you prefer a shallow friend, such as Python, that says yes to (almost) everything we say (at compile time), but is not there for us when we really need it (at run time), or a *true* friend, such as Idris, that tells us that things are not all right all the time, but is there for us when we need it?

Bhaskara at last! ii

- Another way to put it is that “With great power comes great responsibility!”, as the philosopher Ben Parker used to say... Strong typing, and in particular this form of strong typing, that relies on *automated theorem proving* requires some effort from our part in order to precisely tell the compiler how things should be.
- Having said that, let us finish this example by writing `bhask` function.

Bhaskara: first attempt i

- Bhaskara's solution for second-degree polynomials gives no Real solution (when $\delta < 0$), one (when $\delta = 0$), or two (when $\delta > 0$). Since "The Winter is Coming" we should be prepared for two roots:

```
bhask : (a :  Nat) -> (b : Nat) -> (c : Nat)
                                -> (Double, Double)
bhask a b c = ((-b + (sqrt (delta a b c))) / (2 * a),
               (-b - (sqrt (delta a b c))) / (2 * a))
```

- Moreover, we should now work with the Idris `Double` type, because of the `sqrt` function. Run

Bhaskara: first attempt ii

```
*bhask-fun> :t sqrt
sqrt : Double -> Double
```

- Again, our naivete plays a trick on us:

```
Type checking ./bhask-fun.lidr
bhask-fun.lidr:2:19:
  |
2 | > bhask a b c =
          ((-b + (sqrt (delta a b c))) / (2 * a),
           (-b - (sqrt (delta a b c))) / (2 * a))
  |             ^
When checking right hand side of bhask with expected type
        (Double, Double)
```

Bhaskara: first attempt iii

```
When checking an application of function
    Prelude.Interfaces.negate:
        Type mismatch between
                Nat (Type of b)
        and
                Double (Expected type)
```

Load file `bhask-fun.lidr` to see this effect.

- We should write `negate b` instead of `- b`, as `-` is a *binary* operation only in Idris. Moreover, we should *not* be able to negate a natural number! Again, casting is necessary.

- Let us fix all casting problem at once, the final definitions should be as follows:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (cast (b * b)) - (cast (4 * a * c))
bhask : (a :  Nat) -> (b : Nat) -> (c : Nat)
                                 -> (Double, Double)
bhask a b c =
  (negate (cast b) +
    (sqrt (cast (delta a b c))) / cast (2 * a),
   negate (cast b) -
    (sqrt (cast (delta a b c))) / cast (2 * a))
```

Bhaskara: final attempt ii

- We can now play with `bhask`, after executing `:l bhask-fun-fix.lidr`

```
Type checking ./bhask-fun-fix.lidr
*bhask-fun-fix> bhask 1 10 4
(-5.41742430504416, -14.582575694955839) :
                                  (Double, Double)
*bhask-fun-fix> bhask 1 2 3
(NaN, NaN) : (Double, Double)
```

Bhaskara: final attempt iii

- Note that when $\delta < 0$ Idris gives a `NaN` value, which stands for *Not a number*. In other words, `bhask` is **total** as opposed to the **partial** approach in Python where we needed to raise an exception to capture the situation where the roots are not Real numbers.
- Idris can help us identify when a function is total. We simply need to run:

```
*bhask-fun-fix> :total bhask
Main.bhask is Total
```

Wrapping-up i

- First and foremost motivate strong-typing in Idris.
- Introduce notation for functions in Idris. The signature of a function, such as `delta` includes a name, formal parameters and a return type, such as:
  `delta : (a : Nat) -> (b : Nat) -> (c : Nat) -> Int`.
- The formal parameters of a function are declared using the so-called Currying form (after Haskell Curry): currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Wrapping-up ii

- This allows to *partially apply* a function! For instance, we can call `delta 1 2`. This will produce a function that expects a number and then behaves as `delta`.
- Take a look at the following session:

```
*bhask-fun-fix> delta
delta : Nat -> Nat -> Nat -> Int
*bhask-fun-fix> delta 1
delta 1 : Nat -> Nat -> Int
*bhask-fun-fix> delta 1 2
delta 1 2 : Nat -> Int
*bhask-fun-fix> delta 1 2 3
-8 : Int
```

Wrapping-up iii

```
*bhask-fun-fix> (delta 1) 2
delta 1 2 : Nat -> Int
*bhask-fun-fix> ((delta 1) 2) 3
-8 : Int
```

- At the end of the day, `delta 1 2 3` is just *syntax sugar* for `((delta 1) 2) 3`.
- Total functions are such that, for all well-typed inputs, does one of the following:
  - Terminates with a well-typed result
  - Produces a non-empty finite prefix of a well-typed infinite result in finite time We can describe total functions as either terminating or productive.

Wrapping-up iv

  - The halting problem is the difficulty of determining whether a specific program terminates or not, and, thanks to Alan Turing, we know that it's impossible in general to write a program that solves the halting problem.
  - In other words, Idris can't determine whether one of these conditions holds for all total functions. Instead, it makes a conservative approximation by analyzing a function's syntax.

- Type casting. We have used `cast` many times in order to *inject* our values from one type into another.
- Some Read-Eval-Print-Loop (REPL) commands. We have seen how to load a file with `:l`, check its type with `:t`, and check weather a function is total or not with `:total`.

Type-define-refine approach

Type-define-refine approach

The approach is threefold:

1. Type—Either write a type to begin the process, or inspect the type of a hole to decide how to continue the process.
2. Define—Create the structure of a function definition either by creating an outline of a definition or breaking it down into smaller components.
3. Refine—Improve an existing definition either by filling in a hole or making its type more precise.

Following the TDD book Brady17, we use the Atom editor to illustrate the process. (Idris defines an IDE API such that editors like Atom, Emacs or Vi can interact with the REPL.)

The `allLenghts` function i

Let us write a function that given a list of strings computes a list of integers denoting the length of each string in the given list.

- Type

Which should be the type for `allLengths`? Our "problem statement" has already specified it so we just have to write it down:

```
allLenghts : List String -> List Nat
```

After loading the file `tdr.lidr` we get the following.

The `allLenghts` function ii

```
Type checking ./tdr.lidr
Holes: Main.allLenghts
*tdr> allLenghts
allLenghts : List String -> List Nat
Holes: Main.allLenghts
```

There is no surprise with the type but there is `Hole` in our program. Obviously is because we did not declare the equations that define `allLenghts`. This may also occur when Idris fails to type-check a given program.

- Define

The `allLenghts` function iii

Idris may help us think about which cases our function must handle. In the Atom editor, we press Ctrl+Alt+A, producing the following definition:

```
allLenghts : List String -> List Nat
allLenghts xs = ?allLenghts_rhs
```

Of course this is not enough. Here is what Idris says when we load it like this:

```
Type checking ./tdr.lidr
Holes: Main.allLenghts_rhs
```

The `allLenghts` function iv

Let us think about it: what just happened here? Nothing more than create an equation saying that when the `xs` list is given, "something" `?allLenghts_rhs`-ish will happen. Simple but useful when we repeat this process. It is even more useful as a learning tool. Let's continue!

Idris won't leave us with our hands hanging here. It can assist us on thinking about what `?allLenghts_rhs` should look like if we inspect `xs`.

If we press Ctrl+Alt+C on `xs` the editor spits out the following code:

The `allLenghts` function v

```
allLenghts : List String -> List Nat
allLenghts [] = ?allLenghts_rhs_1
allLenghts (x :: xs) = ?allLenghts_rhs_2
```

Two equations were produced because lists in Idris are defined either as the empty list, denoted by `[]`, or a non-empty list denoted by the *pattern* `x :: as`, where `x` is the first element of the given list, which is concatenated to the rest of list in `xs` by the operator `::`.

Nice, and now we have two holes to think about, when the given list is empty and otherwise. Idris allows us to check the type of each hole using the command Ctrl+Alt+T when the cursor is on top of each variable.

The `allLenghts` function vi

```
----------------------------------------
allLenghts_rhs_1 : List Nat

x : String
xs : List String
----------------------------------------
allLenghts_rhs_2 : List Nat
```

- Refine

The `allLenghts` function vii

The refinement of `allLenghts_rhs_1` is trivial: Ctrl+Alt+S (*proof search*) on it gives us `[]`.

For `allLenghts_rhs_2` we need to know however that there exists a `length` operation on strings. We should than apply it `x` and "magically" build the rest of the resulting string. Our code now looks like this:

```
allLenghts : List String -> List Nat
allLenghts [] = []
allLenghts (x :: xs) = (length x) :: ?magic
```

Atom and Idris may help us identify what kind of magic is this. We just have to Ctrl+Alt+T it to get:

```
x : String
xs : List String
---------------------------------------
magic : List Nat
```

So now we need *faith on recursion* (as Roberto Ierusalimschy, a co-author of Lua, says) and let the rest of the problem "solve itself". Finally, we reach the following implementation:

```
> module Main
>
> allLengths : List String -> List Nat
```

The `allLenghts` function ix

```
> allLengths [] = []
> allLengths (x :: xs) = (length x) :: allLengths xs
```

Awesome! For our final magic trick, I would like to know if Idris has a function that given a string produces a list of strings whose elements are the substrings of the first. Try this on the REPL:

```
*type-define-refine/tdr> :search String -> List String
= Prelude.Strings.lines : String -> List String
Splits a string into a list of newline separated strings.

= Prelude.Strings.words : String -> List String
Splits a string into a list of whitespace
```

```
                                         separated strings.
```

```
...
```

It turns out that `words` is exactly what I was looking for! Run the following:

```
*type-define-refine/tdr> :let l = "Here we are, born to be
 we are princess of the universe!"
*type-define-refine/tdr> words l
["Here",
 "we",
 "are,",
 "born",
 "to",
```

The `allLenghts` function xi

```
 "be",
 "kings,",
 "we",
 "are",
 "princess",
 "of",
 "the",
 "universe!"] : List String
```

And Finally

```
*type-define-refine/tdr> :let w = words l
*type-define-refine/tdr> allLenghts w
[4, 2, 4, 4, 2, 2, 6, 2, 3, 8, 2, 3, 9] : List Nat
```

Lab i

In the labs in this short-course you will have to complete or fix some Idris code.

- First lab.

The first lab is to complete the code below using what we have discussed so far.

```
> wordCount : String -> Nat
> -- Type-define-refine this function!
> -- Start by running `Ctrl+Alt+A` to add a definition,
> -- than `Ctrl+Alt+C` to split cases and finally
> -- `Ctrl+Alt+S` to search for proofs(!) that represent
```

Lab ii

```
> -- the code you need! (Intrigued? Ask the instructor
> -- for an advanced course on this topic than = )
>
> average : (str : String) -> Double
> average str =
>          let numWords = wordCount str
>              totalLength =
>                    sum (allLengths (words str))
>          in ?w
> -- Which is the type of `?w1`?
> -- Proof search won't help you here, unfortunately...
> -- Run `:doc sum` at the REPL. Just read the
> -- documentation at the moment, not the type of `sum`.
```

Lab iii

```
>
> showAverage : String -> String
> showAverage str =
>   let m = "The average word length is: "
>       a = average ?w
>   in m ++ show (a) ++ "\n"
> -- Check the type o `w` and think about it!
>
> main : IO ()
> main = repl "Enter a string: " showAverage
```

- Using the example string from above, you should get the following spit at you:

Lab iv

```
Sat Aug 03@18:05:17:type-define-refine$
idris --nobanner tdr.lidr
Type checking ./tdr.lidr
*tdr> :exec main
Enter a string:
Here we are, born to be kings,
  we are princess of the universe!
The average word length is: 3.923076923076923
```

- Moreover, you may *compile it* to an executable with the following command line:

```
idris --nobanner tdr.lidr -o tdr
```

and then execute it, as follows.

Lab v

```
Sun Aug 04@12:39:21:type-define-refine$ ./tdr
Enter a string:
```

The need for dependent types

The need for dependent types

- Overflow conditions in software appear to be a simple thing to implement. An important counter-example is the Ariane 5 rocket that exploded due to a down cast from 64-bit number into a 16-bit one.
  The Ariane 5 had cost nearly $8 billion to develop, and was carrying a $500 million satellite payload when it exploded.

  11 of the most costly software errors in history

- In this chapter we look at a simplified version of the `Vector` datatype, available in Idris' library, to try and understand how *dependent typing* can be useful to have type-safe array handling that could help prevent catastrophes such as the Ariane 5 explosion.

Vector i

- A datatype is nothing but an implementation of some “domain of information”. It could very well represent low level information such as data acquired by a sensor in a Internet of Things (IoT) system or the structure that organizes the decision making process in planning.
- Our datatype here is quite simple but illustrates very well how dependent types may help safe data modeling and implementation.

```
> module Vect
> data Vect : Nat -> Type -> Type where
>     Nil  : Vect Z a
>     (::) : (x : a) -> (xs : Vect k a) -> Vect (S k) a
```

- An array or vector is built or *constructed* using either one of the constructor operations (unary) `Nil` or (binary) `::`. (The `module` keyword here simply defines a *namespace* where `Vect` will live.) After loading this file in Idris you could try

```
*tnfdt> 1 :: Vect.Nil
[1] : Vect 1 Integer
```

at the REPL.

`Vector` iii

- This says that the term `[1]` has type `Vect 1 Integer` meaning that it is a vector with one element and that its elements of the `Integer` type, Idris' basic types.
- Maybe this is a lot to take! *Just breath* and let us think about it for a moment.
- Types are defined in terms of constructor operators. This means that an *instance* of this type is written down as `1 :: Vect.Nil`. In a procedural language you could write it with a code similar to

```
v = insert(1, createVect(1))
```

where `createVect` returns a vector of a given size and `insert` puts an element on the given vector. The point is that we usually create objects or allocate memory to represent data in variables (so called *side effects*) while in functional programming we *symbolically* manipulate them, as in the example above.

- This is a major paradigm-shift for those not familiar with functional programming. Be certain that it will become easier as time goes by, but let's move on!

Dependency i

- Let's look at the instance first and then to the type declaration. Note that the type of `[1]` is `Vect 1 Integer`. The type of a Vect *depends* on its *size*! Think about examples of vectors in programming languages you know. If you query for the type of a given vector, if at all possible, what the run-time of your programming language will answer?
- In Python, for instance, you would get something like,

```
v = [1,2,3]
type(v)
<class 'list'>
```

that is, is a `list` and that's all! In C an array is a pointer! (A reference to a memory address, for crying out loud!)

- In Idris, we know it is a vector and its size, an important property of this datatype. Cool! And so what?
- We can take advantage of that while programming. We could write a function that does *not*, under no circumstances, goes beyond the limits of a vector, that is, index it beyond its range!

The `zip` function i

- The `zip` function simple creates pairs of elements out of two instances of `Vect` *with the same size*. Here is what it look like:

```
> zip : Vect n a -> Vect n b -> Vect n (a, b)
> zip Nil Nil = Nil
> zip (x :: xs) (y :: ys) = (x, y) :: zip xs ys
```

- What on earth is it? Do you remember how to declare a function in Idris? Well, is pretty-much that. The difference here is that we are now programming with *pattern matching*.
- And what is it? Simply define a function by *cases*.

The `zip` function ii

- When we hit an instance of `Vect`, how does it look like? It is either the empty vector, built with constructor `Nil`, or a non-empty vector, built using operator `::`.
- These two cases are represented by each equation above. The first equation declares the case of "zipping" two *empty* vectors and the second one handles two *non-empty* vectors, specified by the *pattern* `x :: xs`, that is, a vector whose first element is `x` and its remaining elements are represented by a (sub)vector `xs`.
- For instance, if we could write

```
*tnfdt> Vect.zip [1,2,3] ["a", "b", "c"]
[(1, "a"), (2, "b"), (3, "c")] : Vect 3 (Integer, String)
```

The `zip` function iii

and get the expected vector of pairs produced by `zip`. (I used `Vect.zip` only because there are other `zip` functions coming from Idris' standard library.)

- Note that the type of `[(1, "a"), (2, "b"), (3, "c")]` is `Vect 3 (Integer, String)` where 3 is the size of the vector and `(Integer, String)`, denoting pairs of integers and strings, is the type of the elements of vector that `zip` calculates.

The `zip` function iv

- Note some additional interesting things about `zip`'s declaration: The signature of `zip` is `zip : Vect n a -> Vect n b -> Vect n (a, b)`. The variable `n` here stands for the size of the vector. Variables `a` and `b` denote the types of the elements of the vectors being zipped.
- That is, the `Vect` type is *generic*, as the type of its elements are underspecified, and is *dependent* on the **number** denoting its size. Again, `n` is a *number*, and `a` (or `b`, for that matter) is a *type*!
- Now, take a look at this:

The zip function v

```
*tnfdt> Vect.zip [1,2,3] ["a", "b"]
(input):1:19-21:When checking argument xs to
  constructor Vect.:::
        Type mismatch between
                Vect 0 a (Type of [])
        and
                Vect 1 String (Expected type)

        Specifically:
                Type mismatch between
                        0
                and
                        1
```

The `zip` function vi

- What does this mean? This is a *type checking* error, complaining about an attempt to zip vectors of different sizes. This is *not* an exception, raised while trying to execute `zip`. This is a *compile* type message, regarding the case of zip a vector of length 1 (the last element of the first vector), and a 0-sized vector (from the second vector).

In Idris, types can be manipulated just like any other language construct.

Conclusion.

Ariane 5 would not have exploded (from the bit conversion perspective) if the function that accidentally cast a 64-bit vector into a 16-bit one was written with this approach.

Wrapping-up

1. Defining datatypes.
2. Defining dependent datatypes.
3. Using dependent datatypes to find errors at compile time.
4. Type expressions.

Insertion sort lab.

Insertion sort lab.

- Here is what we will implement:
- Given an empty vector, return an empty vector.
- Given the head and tail of a vector, *sort* the tail of the vector and then insert the head into the sorted tail such that the result remains sorted.
- At the end, you should be able to run the following at the REPL:

```
*VecSort> insSort [1,3,2,9,7,6,4,5,8]
[1, 2, 3, 4, 5, 6, 7, 8, 9] : Vect 9 Integer
```

I will first walk you through the development of most of the code. At the end of the section I list your activities for this lab.

Type-define-refine i

- Type We will use the `Vect` datatype avaiable in Idris' prelude.

```
> import Data.Vect
```

And it is easy to grasp the signature of our function, so here it goes.

```
insSort : Vect n elem -> Vect n elem
```

- Define Now we add a clause using `Ctrl+Alt+A` on `inSort`, resulting in

```
insSort : Vect n elem -> Vect n elem
insSort xs = ?insSort_rhs
```

and do a case split on variable `xs`.

Type-define-refine ii

```
insSort : Vect n elem -> Vect n elem
insSort [] = ?insSort_rhs_1
insSort (x :: xs) = ?insSort_rhs_2
```

- Refine

```
insSort : Vect n elem -> Vect n elem
insSort [] = []
insSort (x :: xs) = ?insSort_rhs_2
```

- Proof search works just fine for `?insSort_rhs_1` but not so much for `?insSort_rhs_2`, as it simply produces

```
insSort (x :: xs) = ?insSort_rhs_2
```

Type-define-refine iii

- And why is that? Because there is no *silver bullet* and you need to understand the algorithm! The informal specification is quite clear: we need to isert `x` into a sorted (tail) list.

```
insSort (x :: xs) = let l = insSort xs in ?insSort_rhs_2
```

- We can now ask the system to help us with `?insSort_rhs_2` in this context by pressing Ctrl+Alt+L on it. Here is what it creates:

```
insSort_rhs_2 : (x : elem) -> (xs : Vect len elem) ->
                (l : Vect len elem) -> Vect (S len) elem
insSort (x :: xs) =
  let l = insSort xs
  in (insSort_rhs_2 x xs l)
```

Type-define-refine iv

It generates a *stub* of a function with all the variables in the context.

- Since we are following quite easily =(what is going on, we now that we need to rename `insSort_rhs_2` to `insert` (just for readability) and get rid of `xs` in the application, leaving us with

```
insSort (x :: xs) = let l = insSort xs in (insert x l)
```

- Awesome! Let us now define `insert` as the lifting process (with Ctrl+Alt+L) already (overly)defined its type for us. So let us add a clause on `insert`, and case-split `l`. It leaves us with the following code once we search for a proof for hole 1.

Type-define-refine v

```
insert : (x : elem) -> (l : Vect len elem)
                    -> Vect (S len) elem
insert x [] = [x]
insert x (y :: xs) = ?insSort_rhs_2
insSort : Vect n elem -> Vect n elem
insSort [] = []
insSort (x :: xs) = let l = insSort xs in (insert x l)
```

- Proof search will not help us with hole 2, as there are some things we need to figure out. Let us think for a moment what `insert` should do. There are two cases to consider:
- If x < y, the result should be x :: y :: xs, because the result won't be *ordered* if x is inserted after y.

Type-define-refine vi

- Otherwise, the result should begin with y, and then have x inserted into the tail xs.
- In a *type safe* context we need to make sure that `insert` will be able to compare `x` and `y`. In object-oriented terms, that object `x` knows how to answer to message < or that the algebra of `x` and `y` is an order!
- Idris implements the concept of *type classes*, called `interfaces` in Idris and are precisely that: they define operations that a certain datatype must fulfill.
- One such type class is `Ord`.

```
interface Eq a => Ord a where
    compare : a -> a -> Ordering

    (<) : a -> a -> Bool
    (>) : a -> a -> Bool
    (<=) : a -> a -> Bool
    (>=) : a -> a -> Bool
    max : a -> a -> a
    min : a -> a -> a
```

Type-define-refine viii

- It relies on yet another type class called `Eq`, that defines the equality relation and defines a number of operations, including `<`. Type-classes form an important concept in strongly-typed functional programming but we will not explore it any further in this short-course.
- Having said that, we need to constraint `insert` such that `elem` is an *ordered* type.

```
> insert : Ord elem => (x : elem) ->
>                (l : Vect len elem) -> Vect (S len) elem
> insert x [] = [x]
> insert x (y :: xs) = ?insert_rhs
```

Type-define-refine ix

```
> insSort : Ord elem => Vect n elem -> Vect n elem
> insSort [] = []
> insSort (x :: xs) = let l = insSort xs in (insert x l)
```

Lab activities

- So, finally, here is what you should do:

1. Perform all the steps described above until you reach the code above.
2. Replace the meta-variable with the appropriate `if then else` code or search for Ctrl+Alt+M (to generate a `case`-based code) command on the web and try i.t

Programming with type-level functions

Programming with type-level functions

- Here are a couple of examples where first-class types can be useful:
  - Given an HTML form on a web page, you can calculate the type of a function to process inputs in the form.
  - Given a database schema, you can calculate types for queries on that database. In other words, the type of a value returned by a database query may vary *depending on* the database *schema* and the *query* itself, calculated by **type-level functions**.
- This should be useful in a number of contexts such as Data validation in Robotic Process Automation, SQL Injection, (Business) Process Protocol Validation, just to name a few.
- In this section we discuss and illustrate how this way of programming is available in the Idris language.

Formatted output example i

- This examples explores some of the components for the RPA scenario. It exemplifies how to make strings from properly-typed data using type-functions, similarly to the `printf` function in the C programming language.

```
> module Format
>
> data Format =
> Number Format
> | Str Format
> | Lit String Format
> | End
```

Formatted output example ii

- The `Format` datatype is an *inductive* one: is a "list" such that its elements are either `Number`, `Str`, `Lit s` (where `s` is string) or `End`. It will be used to *encode*, or to represent, in Idris, a formatting string.
- Try this at the REPL:

```
*pwfct> Str (Lit " = " (Number End))
Str (Lit " = " (Number End)) : Format
```

- This instance of `Format` represents the formatting string "%s = %d" in C's `printf`.
- So far, nothing new, despite the fact that we now realize that our datatypes can be recursive.

Formatted output example iii

- Function `PrintfType` is a *type-level function*. It describes the *functional type* associated with a format.

```
> PrintfType : Format -> Type
> PrintfType (Number fmt) = (i : Int) -> PrintfType fmt
> PrintfType (Str fmt) = (str : String) -> PrintfType fmt
> PrintfType (Lit str fmt) = PrintfType fmt
> PrintfType End = String
```

- Recall that a functional type is built using the `->` constructor. The first equation declares that a `Number` format is denoted by an `Int` in the associated type. The remaining equations define similar denotations.

Formatted output example iv

- Try this at the REPL:

```
*pwfct> PrintfType (Str (Lit " = " (Number End)))
String -> Int -> String : Type
```

- As I mentioned before, the format `(Str (Lit " = " (Number End)))` encodes the C formatting string “%s = %d”. The functional type that denotes it is `String -> Int -> String`, that is, a function that receives a string and an integer and returns a string.

Formatted output example v

- Again, PrintfType is a type-function, that is, it defines a type. Of course, we can use it to specify, for instance, the return type of a function. The recursive function printfFmt receives a format, a string and returns a term of PrintfType that *depends on the format given as first argument*!

```
> printfFmt : (fmt : Format) ->
>             (acc : String) -> PrintfType fmt
> printfFmt (Number fmt) acc =
>           \i => printfFmt fmt (acc ++ show i)
> printfFmt (Str fmt) acc =
>           \str => printfFmt fmt (acc ++ str)
> printfFmt (Lit lit fmt) acc =
```

Formatted output example vi

```
>              printfFmt fmt (acc ++ lit)
> printfFmt End acc = acc
```

- Function `toFormat` is a normal function that transforms a string denoting a format and creates a *type* `Format`. Function `printf` is defined next.

```
> toFormat : (xs : List Char) -> Format
> toFormat [] = End
> toFormat ('%' :: 'd' :: chars) = Number (toFormat chars)
> toFormat ('%' :: 's' :: chars) = Str (toFormat chars)
> toFormat ('%' :: chars) = Lit "%" (toFormat chars)
> toFormat (c :: chars) =
```

Formatted output example vii

```
>   case toFormat chars of
>       Lit lit chars' => Lit (strCons c lit) chars'
>       fmt => Lit (strCons c "") fmt
> printf : (fmt : String) ->
>          PrintfType (toFormat (unpack fmt))
> printf fmt = printfFmt _ ""
```

- Try this out at the REPL:

```
*pwfct> :let msg =
        "The author of %s, published in %d, is %s."
*pwfct> :let b = "A Brief History of Time"
*pwfct> :let a = "Stephen Hawking"
*pwfct> :let y = the Int 1988
```

Formatted output example viii

```
*pwfct> printf msg b y a
"The author of A Brief History of Time,
 published in 1988, is Stephen Hawking." : String
```

- At this point you should be able =(to understand what is going on. Why does `printf` takes four arguments? Shouldn't it be just one? (The `fmt : String` above.)
- For variable `y` we had to make sure it is an `Int` (finite), not an `Integer` (infinite) number, due to `PrintfType` definition. This is what `the Int 1988` does. Try it without the casting and see what happens. . .

Conclusion

- The point here is that we can use types to help organize the world.
- Recall the SQL Injection example from the introductory section. The problem there was the fact that everything was a string.
- Using the concepts discussed here we could type information coming from forms and check them before sending them to the DBMS!

Caveats i

(From TDD book.)

- In general, it's best to consider type-level functions in exactly the same way as ordinary functions. This isn't always the case, though. There are a couple of technical differences that are useful to know about:
- Type-level functions exist at *compile* time only. There's no runtime representation of Type, and no way to inspect a Type directly, such as pattern matching.

Caveats ii

- Only functions that are total will be evaluated at the type level. A function that isn't total may not terminate, or may not cover all possible inputs. Therefore, to ensure that type-checking itself terminates, functions that are not total are treated as constants at the type level, and don't evaluate further.

Infinite data and processes

Infinite data i

- Streams are infinite sequences of values, and you can process one value at a time.
- When you write a function to generate a Stream, you give a prefix of the Stream and generate the remainder recursively. You can think of an interactive program as being a program that produces a potentially infinite sequence of interactive actions.

```
> %default total
> data InfIO : Type where
>     Do : IO a -> (a -> Inf InfIO) -> InfIO
> (>>=) : IO a -> (a -> Inf InfIO) -> InfIO
> (>>=) = Do
> loopPrint : String -> InfIO
> loopPrint msg = do putStrLn msg
>                    loopPrint msg
> partial
> run : InfIO -> IO ()
> run (Do action cont) = do res <- action
>                           run (cont res)
```

Infinite data iii

- Try the following at the REPL:

```
:exec run (loopPrint "on and on and on...")
```

and a non-terminating execution will present itself. As expected, `run` is *not* total:

```
*streams/streams> :total run
Main.run is possibly not total due to recursive path:
    Main.run, Main.run
```

- The type `InfIO`, as the name suggests, is a type of infinite IO actions, denoted by the type variable `a`. The `Do` constructor receives an IO action and produces an infinite IO action, by recursion.

- Function `loopPrint` is one such *action generator*.
- Let us take this slowly: First of all, what is the `Inf` type?

```
Inf : Type -> Type
Delay : (value : ty) -> Inf ty
Force : (computation : Inf ty) -> ty
```

- `Inf` is a generic type of potentially infinite computations.
- `Delay` is a function that states that its argument should only be evaluated when its result is forced.
- Force is a function that returns the result from a delayed computation.

Another example with infinite data i

- `InfList` is similar to the List generic type, with two significant differences:
  - There's no `Nil` constructor, only a `(::)` constructor, so there's no way to end the list.
  - The recursive argument is wrapped inside `Inf`.

```
> data InfList : Type -> Type where
>     (::) : (value : elem) -> Inf (InfList elem) ->
>     InfList elem
```

- Function `countFrom` is an example on how to use `Inf`.

Another example with infinite data ii

```
> countFrom : Integer -> InfList Integer
> countFrom x = x :: Delay (countFrom (x + 1))
```

The Delay means that the remainder of the list will only be calculated when explicitly requested using Force.

Try the following at the REPL:

```
*streams> countFrom 0
0 :: Delay (countFrom 1) : InfList Integer
```

Streams i

- Idris has streams in its prelude.

```
data Stream : Type -> Type where
   (::) : (value : elem) -> Inf (Stream elem) ->
                            Stream elem
repeat : elem -> Stream elem
take : (n : Nat) -> (xs : Stream elem) -> List elem
iterate : (f : elem -> elem) -> (x : elem) -> Stream elem
```

- Execute

```
 (iterate (+1) 0)
*streams/streams> (iterate (+1) 0)
0 ::
Delay (iterate (\ARG => prim__addBigInt ARG 1) 1) : Stream
```

and try to grasp which type is this.

- Here are some cool stuff we can do with streams, try it out:

```
Idris> take 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : List Integer
```

The syntax [1..] generates a Stream counting upwards from 1.

- This works for any countable numeric type, as in the following example:

Streams iii

```
Idris> the (List Int) take 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : List Int
```

or

```
Idris> the (List Int) (take 10 [1,3..])
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19] : List Int
```

- Now, which is the relationship between all this machinery and the motivation presented at the beginning of the course?
  - Are there any relations among IOT sensors and streams?
- You should probably have realized by now that `run` is an *infinite process* executing on an *infinite stream* of data!

Making infinite processes total i

- As trivial as it may sound, a way to make a function terminate is simply to define a “time out”.
- In the following example, this is denoted by the `Fuel` datatype. The `Lazy` datatype is similar to the `Inf` we have seen before, it “encapsulates” infinite data and only computes it when necessary.

```
> data Fuel =
>  Dry | More (Lazy Fuel)
>
> tank : Nat -> Fuel
```

Making infinite processes total ii

```
> tank Z = Dry
> tank (S k) = More (tank k)
>
> partial
> runPartial : InfIO -> IO ()
> runPartial (Do action f) =
>            do res <- action
>               runPartial (f res)
>
> run2 : Fuel -> InfIO -> IO ()
> run2 (More fuel) (Do c f) =
>      do res <- c
>         run2 fuel (f res)
```

Making infinite processes total iii

```
> run2 Dry p = putStrLn "Out of fuel"
>
> partial
> main : IO ()
> main = run2 (tank 10) (loopPrint "vroom")
```

Inf vs. Lazy i

- If the argument has type Lazy ty, for some type ty, it's considered smaller than the constructor expression.
- If the argument has type Inf ty, for some type ty, it's not considered smaller than the constructor expression, because it may continue expanding indefi- nitely. Instead, Idris will check that the overall expression is productive

Domain-specific commands lab.

Domain-specific commands lab.

In this lab we will restrict interactive program's to perform only specific actions as opposed to the stream example.

Commandtype i

- Type `Command` defines an interactive interface that `ConsoleIO`, a type that describes interactive programs that support only reading from and writing to the console, programs can use.
- You can think of it as defining the capabilities or permissions of interactive programs, eliminating any unnecessary actions.

```
> data Command : Type -> Type where
>     PutStr : String -> Command ()
>     GetLine : Command String

> data ConsoleIO : Type -> Type where
```

Commandtype ii

```
>     Quit : a -> ConsoleIO a
>     Do : Command a -> (a -> Inf (ConsoleIO b)) -> Console

> (>>=) : Command a -> (a -> Inf (ConsoleIO b)) -> ConsoleI
> (>>=) = Do
```

DSL i

- A domain-specific language (DSL) is a language that's specialized for a particular class of problems. DSLs typically aim to provide only the operations that are needed when working in a specific problem domain in a notation that's accessible to experts in that domain, while eliminating any redundant operations.
- In a sense, ConsoleIO defines a DSL for writing interactive console pro- grams, in that it restricts the programmer to only the interactive actions that are needed and eliminates unnecessary actions such as file processing or net- work communication.

The lab activities

- Your mission, should you choose to accept it, is to understand and execute program `ArithCmd.idr` from Chapter 11 of the TDD book, also available at this short-course repo.